PATENT
Attorney Docket No. P4803
Client/Matter No. 80168.0097
Express Mail No. EL700671813US

SYSTEM FOR INTERFACING AN APPLICATION PROGRAM WITH DIVERSE DATABASES

BACKGROUND OF THE INVENTION

Technical Field

The present invention relates generally to database query methods, and more particularly, to a system for interfacing an application program with more than one database instance.

Statement of the Problem

Queries to a database from an application program have historically been written with a specific database instance (data model) in mind. The application is thus typically tailored to generate database query commands of the form required for the specific database. Since each database model has its own set of data query/access commands, the application program must be re-coded and re-assembled in order to access a database other than the one with which it was originally designed to interface. Therefore, not only is it time consuming and costly to re-program an application to provide access to a different database, the application must also be re-tested after the new database query commands have been retrofitted into the application. The testing procedure is also time consuming, and it may be difficult to ensure that the revised application is as robust as the original application without running the application in its real-world environment, a situation which may entail a certain amount of undesirable risk.

There are a number of reasons why application programs are tailored for a specific type of database. These reasons include performance considerations as well as the fact that such tailoring has simply been the 'normal' way to program a database-related application. Furthermore, since

the structure of the database is typically known in advance, it is seemingly 'natural' to structure the database queries in accordance with the structure of the database and known features of a particular database query language. It has heretofore also been assumed that acceptable performance of most applications utilizing database queries could be achieved only by using 'built-in' queries tailored specifically for the database's particular query language. Therefore, when it is required that an application be run with a database other than the one for which the application was originally intended, the application must be re-programmed. Re-programming is time consuming and requires that the modified application program also be re-tested. Re-testing of the application not only requires further time and effort, but also introduces an additional risk of having undetected errors present in the application when it is introduced back into the user environment.

Thus, a mechanism is needed which enables a given application program to run with any type of database and associated query language without incurring the overhead and risk associated with re-programming and re-testing the application for each different database type.

Solution to the Problem

The present invention overcomes the aforementioned problems of the prior art and achieves an advance in the field by providing a method for interfacing an application program with a number of different databases without necessitating re-programming of the application for each different database. The present system provides a mechanism, transparent to the application program, which handles I/O requests from the application in such a manner as to allow the application to interface with any desired database, without re-programming or otherwise changing the specific format of the application to accommodate the particular query format required by a given database.

In accordance with the present system, query strings comprising queries used by an application program are loaded from a text file into a

query lookup table. The application then utilizes a database interface query function to access the queries in the lookup table. The query function uses the query name provided by the application to locate the corresponding query string in the lookup table. The query function then performs parameter substitution on the query string and sends the query to the database. The query results are then retrieved, formatted, and sent to the querying application.

The present system allows an application program to access data stored on databases organized by various database management systems which employ different query formats. Although the databases organize the data differently, the same content may be extracted by the present system. Changes to the database queries take place outside the application code in a simple text file that is loaded by the querying application. Testing of the changes is isolated to ensuring the queries return the expected data.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a diagram illustrating interrelationships between elements of the present system for managing database queries;

Figure 2 is a flowchart illustrating preliminary steps which are performed in the development process prior to operation of the present system;

Figure 3 is a flowchart illustrating operation of the present system;

Figure 4 is a diagram illustrating elements of the present system which facilitate accessing of two different databases; and

Figure 5 is a diagram illustrating operation of the present system in an object-oriented environment.

DETAILED DESCRIPTION

Figure 1 is a diagram illustrating interrelationships between elements of the present system 100 for managing database queries. As shown in Figure 1, query strings comprising database queries used by application program 101 are stored in a query text file 103. Queries are read from

query text file 103 by database interface query function 102 and loaded into a query lookup table 105. Database interface function 102 uses queries stored in lookup table 105 to access database 104 in response to a database access request by application 101.

Figure 2 is a flowchart illustrating steps which are performed in the development process prior to operation of the present system. As shown in Figure 2, the development process can be split into two separate phases, one phase, beginning at step **201,** for development of application 101 which requires database access, and the other phase, beginning at step **202,** for development of one or more database instances D1–Dn, each of which is compatible with application 101. Initially, at step **201**, the data requirements for application 101 are defined at step **205**. This step includes defining the types of database access required by the application, such as, where in the application the calls to the database will be made, and what database access is required (e.g., update, search, etc.). The development process for database D1, or multiple databases (D2–Dn), with which application 101 will interface, may take place in parallel with development of application 101. The database development is shown as starting at step **202**, as indicated by the blocks with suffixes (D1) and (Dn), on the right side of Figure 2, where 'D1' indicates database number 1, and 'Dn' indicates database number *n*, with any number (n-2) of databases in between.

On the application development (left) side of Figure 2, at step **210**, a data model, corresponding to the data requirements of application 101, is created. Application 101 is then created in accordance therewith. At step **215**, a database interface function 102 is created that will accept data from database 101 and perform the appropriate formatting for use by the rest of the application 101. The present system implements both application 101 and database interface function 102 as objects. Steps 205, 210, and 215 need be performed only once, i.e., at the time application 101 is written.

On the database development (right) side of Figure 2, at steps **211(1)–(n)**, independent of the development of application 101 (i.e., developed asynchronously with respect thereto), a data model is created for each

database with which it is desired to interface application 101. A corresponding database instance 104 [DB1-DBn] is then created for each database data model. At steps **220(1)-(n)**, the database queries are written that will be used for extracting the desired data from each database 104 [DB1-DBn]. Clearly, the queries must be written so that the data returned by each query matches the data expected by the database interface objects. Next, at steps **225(1)-(n)**, the queries written in step 220 are stored in a query text file 103(1)-(n). Steps 211, 220 and 225 are thus performed for each different database DB1-DBn with which application 101 will interface; for example, a custom-designed database for a first group of users, and perhaps an existing database for a second user group. The use of multiple query text files 103 by the present system 100 is further described below with respect to Figure 4.

Finally, at step **230**, development of application 101 and database DB1 (or databases DB1-DBn) is complete, and operation of system 100 proceeds as described below with respect to Figure 3.

Figure 3 is a flowchart illustrating operation of the present system in an exemplary embodiment. As shown in figure 3, prior to execution of the first database access request by application 101, at step **305**, database interface function 102 reads query text file 103. Next, at step **310,** query text file 103 is loaded into a query lookup table 105. Queries are stored as strings, by query name, in the lookup table. Steps 305 and 310 may be performed upon initial execution of database interface function 102, or, alternatively, on demand while application 101 is running. In the present embodiment, text file 103 is loaded into query lookup table 105 as a hash table, but could be loaded, alternatively, in any format compatible with the operation of database interface function 102. Database interface function 102 then waits, at step **315**, for a database access request from application 101.

At step **320**, application 101 makes a database access request, invoking database interface function 102, which controls access to the query strings. The query name and the arguments required by the specific query

are provided by the application, accompanying the call to database interface function 102. These arguments are typically either search criteria or data used to update the database content. Application 101 then waits, at step **325**, for a response back from database interface function 102.

At step **330**, database interface function 102 receives the database access call from application 101. Database interface function 102 then performs a query lookup by name in query lookup table 105 at step **335**, and at step **340** performs textual substitution of the parameters (provided by the application) into the positions identified in the query string. At step **345**, the query is submitted to database 104. Database interface function 102 passes the query string to database 104 using a mechanism such as JDBC or other API for communicating queries. Next, at step **350**, a response from the database is received, which is formatted by database interface function 102 into either records or objects in accordance with the type of format expected by application 101. Finally, at step **355**, the resulting records or objects are returned to application 101.

Figure 4 is a diagram illustrating the manner in which the present invention provides for an interface between a given application and dissimilar databases. As shown in Figure 4, queries used by application program 101 to access database DB1 [ref. no. 104(1)] are stored in query text file 103(1). Query text file 103(1) is then read by database interface function 102, and the queries are stored (after being hashed) in query lookup table 105(1). Next, when application program 101 issues a request for access to database DB1, database interface function 102 locates, by name, the corresponding query string stored in query lookup table 1 [ref. no. 105(1)] which is specifically formatted in accordance with the syntax of the DBMS (for example, Oracle DBMS) used to implement DB1. Database Interface Function 102 then performs parameter substitution on the query string and sends the query to database DB1. Database interface function 102 then retrieves the query results, formats the results, and sends the formatted results to application 101. The operation described thus far with respect to Figure 4 is exactly as described above with respect to Figure 3.

Figure 4 further shows the procedure employed by the present system when it is desired to have application program 101 access data stored on a database DB2 (ref. no. 104(2)) managed by a DBMS (e.g., Sybase) that is different than the DBMS used with DB1. In this case (as shown in Figure 2 at step 220), each of the queries stored in text file 103 is modified for syntax compliance with DB2. Next (at step 225 in Figure 2), the queries are stored in text file 103(2). Database Interface Function 102 next reads query text file 103(2), then hashes and stores the queries therein in query lookup table 105(2), as indicated by arrow 401. In the present example, when application 101 issues a database access request, database interface function 102 locates the corresponding query string stored in query lookup table 105(2), which has effectively replaced query lookup table 105(1). Database Interface Function 102 then performs parameter substitution on the query string and sends the query to database DB2, as indicated by arrow 402. Finally, database interface function 102 retrieves the query results from DB2, formats the results, and sends the formatted results to application 101.

Therefore, it can be seen that the underlying queries stored in any given text file 103 may be changed to accommodate access between application 101 and any desired database (e.g., DB1 or DB2) or any new table/view definitions without changing any of the underlying compiled code in application 101. An alternative embodiment of the present system may be coded with a switch to allow application program 101 to access either DB1 or DB2. In this situation, query lookup tables 105(1) and 105(2) are advantageously pre-populated with hashed queries from text files 103(1) and 103(2) prior to operation of application 101.

The method of the present invention is further explained with reference to two hypothetical databases. For the purpose of the present example, these two databases are database DB1 [ref. no. 104(1)] and database DB2 [ref. no. 104(2)], as shown in Figure 4. A simple collection of data about an individual is employed in the present example to demonstrate the mechanism of the present invention for handling database queries. The information stored in each of the two databases (DB1 and

DB2) shown below includes a person's name and some contact information for the person. The first data model assumes only a single set of contact information for an individual, while the second data model allows multiple sets of contact information.

For the first database, assume that the following information is stored in a single table called *Person*:

Person [DB1]

- id (PersonID)
- surname (SurName)
- given name (GivenName)
- title (Title)
- phone number (PhoneNumber)
- email address (EmailAddress)

A query to access this information could be formatted as follows:

```
SELECT PersonID, Title, GivenName, SurName, PhoneNumber,
        EmailAddress
FROM Person WHERE PersonID={0};
```

The {0} in the above query is a replaceable parameter which allows the application to request information on a specific person. Multiple parameters may be used in the general case. The above query returns the requested person's primary contact information.

The second hypothetical database of the present example, DB2, contains the same information as the first database above. However, this second database is organized in a slightly different fashion. This second database has two tables, *Person* and *ContactInfo*:

Person [DB2]

- id (PersonID)
- title (Title)
- first name (FirstName)
- last name (LastName)
- middle name (MiddleName)

ContactInfo [DB2]

- id (PersonID)
- type (Type)
- phone number (PhoneNumber)
- email address (EmailAddress)
- street address, first line (Street1)
- street address, second line (Street2)
- city (City)
- state (State)
- zip (ZipCode)

The *ContactInfo* table allows multiple records for one person, with the different records distinguished by type. Assume, in this example, that the type of interest is '1', which corresponds to the person's primary contact information. A query to extract the same information from the second database DB2 could be formulated as:

```
SELECT a.PersonID, a.Title, a.FirstName, a.LastName,
      b.PhoneNumber, b.EmailAddress
FROM Person a, ContactInfo b
WHERE a.PersonID=b.PersonID
AND a.PersonID={0} AND b.Type=1;
```

For the present example, assume that there is a need to select contact information from the database. In this case, the discriminator (name of the query) is *Person.SelectContactInfo*. The query definitions for the two databases would then have the following formats:

DB1

Person.SelectContactInfo=SELECT PersonID, Title, GivenName, SurName, PhoneNumber, EmailAddress FROM Person WHERE PersonID={0}

DB2

Person.SelectContactInfo=SELECT a.PersonID, a.Title, a.FirstName, a.LastName, b.PhoneNumber, b.EmailAddress FROM Person a, ContactInfo b WHERE a.PersonID=b.PersonID AND a.PersonID={0} AND b.Type=1

Note that for both cases above, the information returned (in the same order) is:

an identifier (PersonID in both databases), Title, First name (FirstName vs GivenName), Last name (LastName vs SurName), Phone number, and Email Address. Note also that the argument passed ({0}) references the identifier in both cases.

The two databases described above organize the data differently, but the same content may be extracted. The change to the query takes place outside the application program code in a simple text file that is loaded by the application. As long as the same data can be retrieved from the two databases, the present system provides a mechanism for its retrieval from either database without necessitating any modifications to the underlying application.

Figure 5 is a diagram illustrating operation of the present system 100 in an object-oriented environment. In the current embodiment (an object-oriented programming environment), a parent class (501) includes the query function (shown as database interface function in Figure 1) and a child class (subclassed object) 505 formats the results of a query, as explained in detail below. Business objects (506) provide the structure to carry the database request discriminator and parameters to the database interface function, and to carry the return data from the database interface function back to the application.

The parent class is written to handle the common application logic described for database interface function 102. The parts that are common to all queries are:

(1) loading queries into the lookup table (steps 305 and 310 in Figure 3);

(2) performing query lookup (step 335); and

(3) performing parameter substitution (step 340)

The static data required (in query lookup table 105) can be a class attribute, available to all subclasses without requiring reloading for each query. Subclasses only override the portions of the processing required,

which is, most frequently, only the portion that formats the response from database 104.

Although a procedural implementation could perform the loading of the file into a common data area, the handling of the specific requests would require more complicated coding using conditionals or other decision making mechanisms to direct the query submission and response to the proper procedures.

The method of the present system 100 is described below with reference to Figure 5, in conjunction with Tables 1 and 2 (below). Initially, text file 103 containing query text strings 504 is loaded into query lookup table 105 (*sqlProperties*), which is an attribute of parent class 501. The actual lookup in the hashtable (or other lookup table) and format method is performed by *getMessageFormat*: [Table 1, lines 40 - 44], which uses *getProperty* [Table 1, lines 47 - 51] to actually retrieve the query string, the name of which has an exemplary format of the form:

<class name>.<discriminator>

The operation of the present system 100 is explained below with continuing reference to Figure 5, and also with reference to Tables 1 and 2, using the following query 504(1) for retrieving personal contact information as an example:

<u>Query 504(1)</u>

Person.SelectContactInfo=SELECT PersonID, Title, GivenName, SurName, PhoneNumber, EmailAddress FROM Person WHERE PersonID={0}

Next, in an exemplary embodiment, an object is created (*BusinessObject* 506) which is a carrier object that contains the two required data items (discriminator and list of arguments to the query). This object becomes a member (*businessObject)* of the appropriate subclass of *DBObject*. Once the database request has been issued and the results processed, a new instance (*returnObj*) of business object 506 is created to hold the result data. *BusinessObject* 506 is both a carrier of data to the database 104, as well as a carrier of data returning from the database.

In the present example, a *DBObject* subclass is created that properly formats the resulting data sets from such a query (i.e., the data values of *id*, *title*, *first name, last name, phone number*, and *email address*). A carrier object (e.g., *businessObject* 506) is also created that another layer of the application sends to the database 104. The *businessObject* 506 looks (from an abstract viewpoint) like a record containing a pair of items, a discriminator (e.g., *Person.SelectContactInfo*), and a list of arguments (e.g., {*id*}). Specifically, the *businessObject* 506 containing the discriminator *Person.SelectContactInfo* is created, and an argument list of one item, the person's user id {"jsbach"}. This name/value (discriminator/argument list) 'pair' is sent to the *DBPerson* subclass 505 [Table 2] of DBObject by calling the *doTransaction* method (shown as one of parent class operations 503) of the DBObject [Table 1, lines 13 - 19].

Next, *doTransaction* calls *getDBMessageFormat* [Table 1, lines 40 - 44], which retrieves the query by lookup in the table (105) of name/value pairs, resulting in retrieval of the query 504(1), shown above. The query parameters are replaced with the arguments (*args*) provided by the call *fmt.format* [Table 1, line 18]. *DBMessageFormat* is a utility class that scans the input String for arguments of the form {0}, {1}, etc, takes arguments from the passed in array of arguments, and substitutes them by position. For example, a reference of {0} will be replaced by the 0th argument of the argument array - *args*[0]. *DBMessageFormat* is an extension of the standard Java™ class *MessageFormat*. The result of the use of DBMessageFormat is substitution of the arguments into the query string, providing the specific query that will be sent to the database:

> *Person.SelectContactInfo*=SELECT PersonID, Title, GivenName, SurName, PhoneNumber, EmailAddress FROM Person WHERE PersonID='jsbach'

Note the one replacement location indicated by {0} in the query as stored in the lookup tables is replaced by the actual argument, "jsbach".

Now that this query is fully specified for the database of interest, *doTransaction* calls the *execute* method [Table 1, line 18], which makes the

actual connection to the database and sends the query to the database. The two parts actions resulting in the call to the database are *createStatement* and *stmt.execute* [Table 1, lines 26 - 27]. The result may be a set of rows, retrieved into *resultSet* [Table 1, line 29], or a count of rows updated, stored in an attribute of *returnObj* [Table 1, lines 32 - 34].

When a *resultSet* is returned from the database, *processResultSet* 507 [Table 1, lines 53 - 57; overridden in Table 2] extracts the data. At this point, iteration is performed over the rows returned to *resultSet* to make a new result record. The record is filled with field values from the row, and the result record is stored in the *returnObj* [Table 1, line 29].

Next, the data is stored in a simple object which is essentially a data record. To allow for more than one row of data to be returned, the *returnObj* (which is an instance of *businessObject* 506) holds the resulting record or records in an array. Application logic can then extract whatever elements are needed from this array one at a time. The resulting data record from the example query contains the following information:

- jsbach
- Kapellmeister
- Johann
- Bach
- 123-456-7890
- jsbach@brandenburg.org

Finally, the query results are returned (via *returnObj*) to the caller.

Actual Java™ code is listed below in **Table 1** for the *DBObject* parent class that, in accordance with an exemplary embodiment of the present method, performs a translation between a database access request in a given application and the database of interest.

Table 1 *DBObject* parent class

```
/* DBObject.java */
public abstract class DBObject {

    /** Hashtable containing queries. */
    private static Properties sqlProperties;

    /** Business object we wrap. */
    protected BusinessObject businessObject;

    /** JDBC database connection. */
    protected Connection connection;

    /** Do transaction with database. */
    public BusinessObject doTransaction() throws SQLException {
        DBMessageFormat fmt =
            getMessageFormat(businessObject.getDiscriminator());

        return execute(fmt.format(businessObject.getArgs()));
    } // doTransaction


    /** Execute a SQL Statement. */
    protected BusinessObject execute(String sqlString) throws SQLException {
        BusinessObject returnObj = null;

        Statement stmt = connection.createStatement();
        if (stmt.execute(sqlString)) {
            // was select; do result set
            returnObj = processResultSet(stmt.getResultSet());
        } else {
            // was insert/update/delete; do row count
            returnObj = (BusinessObject)businessObject.getClass().
                newInstance();
            returnObj.setResultCount(stmt.getUpdateCount());
        }

        return returnObj;
    } // execute

    /** Get DBMessageFormt statement. */
    protected DBMessageFormat getMessageFormat(String stmtName) {
        String stmt = getProperty(getClass(), stmtName);
        return new DBMessageFormat(stmt);
    } // getMessageFormat
```

```
    /** Get property related to specific class. */
    private String getProperty(Class objClass, String propName) {
        return sqlProperties.getProperty(objClass.getName() +
                                         "." + propName);
    } // getProperty(Class, String)

    /** Fill in the BusinessObject with the data from the result set. */
    protected BusinessObject processResultSet(ResultSet rs)
        throws SQLException {
        return null;
    } // processResultSet

} // DBObject
```

End of Table 1

Table 2 contains exemplary Java™ code that represents a sample child class 505 for a Person entity. It shows the key elements and what is normally overridden in the child classes. The parent class code that is overridden is the code that retrieves the data from *resultSet*. The child classes usually override only *processResultSet*, although they can override any methods that are not designated as final. Queries for different databases are written to ensure that the data returns in the same order.

<u>**Table 2** *DBPerson* child class</u>

```
/ * @(#)DBPerson.java
 *
 * DBObject wrapper around BusinessObject "Person".
 */
public class DBPerson extends DBObject {
 /**
    * Parses result set, creates a business object.
    * @param rs Result Set
    * @return business object
    * */
   public BusinessObject processResultSet(ResultSet rs) {
            // get data out of ResultSet
      if (rs != null) {
         obj = new Person();
         while (rs.next()) {
            String id = rs.getString(1);
            String lastName = rs.getString(2);
            String firstName = rs.getString(3);
            String middleName = rs.getString(4);
            int role = rs.getInt(5);
            PersonData d =
               new PersonData(id, "", firstName, middleName,
                        lastName, role);
            obj.addDataObject(d);
      }
      return obj;
   } // processResultSet
} // DBPerson
```

<u>End of Table 2</u>

While exemplary embodiments of the present invention have been shown in the drawings and described above, it will be apparent to one skilled

in the art that various embodiments of the present invention are possible. For example, the specific Java™ code listed in Tables 1 and 2, as well as other specific instances of code set forth in this document, should not be construed so as to limit the embodiments described herein. Furthermore, although the disclosed embodiments are described with respect to an object-oriented programming environment, the system disclosed herein may, alternatively, be implemented using procedural programming techniques. Modification may be made to these and other specific elements of the invention without departing from its spirit and scope as expressed in the following claims.